Java Persistence API

Agenda

1. Introduction
Entities
Entity Relationship
Cascade Operation
Inheritance
Managing Entities
Entity LifeCycle Management
API Operations on Entity
Persistence Unit
Spring Configuration
Querying Entities
Difference Between JPA and Hibernate
Resources

Introduction

- ***The Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications.***
- ***JPA itself is just a specification, not a product, it cannot perform persistence or anything else by itself.***
- ***JPA is just a set of interfaces, and requires an implementation. There are open-source and commercial JPA implementations to choose from and any Java EE 5 application server should provide support for its use.***
- ***JPA also requires a database to persist to*.**

Java Persistence

The Java Persistence consists of four areas:

- ***The Java Persistence API***
- ***The Query Language***
- ***The Java Persistence Criteria API***
- ***Object/Relational Mapping Metadata***

Entities

- ***An entity represents a table in a relational database, and each entity instance corresponds to a row in that table.***
- ***An entity is a lightweight persistence domain object.***
- ***The primary programming artifact of an entity is the entity class***

Requirements For Entity Classes

- *The class must be annoyed with java.persistence.Entity annotation .*
- *The class must not be declared final.*
- *The class must implement the Serializable interface.*
- *Entities may extend both entity and non-entity classes, and non-entity classes may extend entity class.*
- *Persistent instance variable must be declared private, protected, or package-private.*

For Example:

```
@Entity
class Employee{
….
}
```

and Properties

The following Collections are used:

- ***java.util.Collection***
- ***java.util.Set***
- ***java.util.List***
- ***java.util.Map***

Entity Field and Properties[Conti..] Properties [Conti..]

If a Field or Property of an entity consists of a collection of basic types or embeddable classes use the java.persistence.ElementCollection annotation on the field or property

The 2 attributes of @ElementCollections *are:*

- ***fetch***
- ***targetClass***

Entity Field and Properties[Conti..]

The fetch attribute is used to specify whether the collection should be retrieved lazily or eagerly, using java.persistence.FetchType contacts of **LAZY** *or* **EAGER**, *respectively. By default, the collection will be fetched* **LAZY.**

The following entity, Employee, has a persistent field, first name, which is collection of String classes that will be fetched eagerly. The targetClass element is not required, because it uses generics to define the field.

```
@Entity
 public class Employee{
 ….
 @ElementCollection(fetch=EAGER)
  protected Set<String> first name = new HashSet();
 ….
 }
```

Primary Keys in Entities

Each entity has a unique object identifier.

A Primary key class have the following requirements:

- *The access control modifier of the class must be* public.
- *The properties of the primary key class must be* public or protected *if property-based access is used.*
- *The class must have a* public *default constructor and must be serializable.*
- *The class must implement the* hashCode() *and* equals(Object other) *methods.*

```
@Entity
public class Employee {
@Id @Generated Value
private int id;
private String first name;
….
public int getId()  {  return id;  }
public void setId(int id)  {  this.id = id;  }
….
}
```

Primary Key

M

There are four types of relationship multiplicities:

- ***@OneToOne***
- ***@OneToMany***
- ***@ManyToOne***
- ***@ManyToMany***

The direction of the relationship can be:

- ***bidirectional***
- ***unidirectional***

OneToOne Mapping

Each entity instance is related to a single instance of another entity

For Example:

```
@Entity
  public class Employee {
@Id
@Column(name="EMP_id")
private long id;
....
@OneToOne(fetch=FetchType.LAZY)
@JoinColumn(name="ADDRESS_ID")
 private Address address;
....
}
```

OneToMany Mapping

An entity instance can be related to a multiple instance of other entities.

For Example:

```
@Entity
  public class Employee {
@Id
@Column(name="EMP_id")
private long id;
….
@OneToMany(mappedBy="owner")
private List<Phone> phones;
….
}
```

ManyToOne Mapping

Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship.

For Example:

```
@Entity
 public class Phone{
@Id
private long id;
….
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="OWNER_ID")
private Employee owner;
….
}
```

ManyToMany Mapping

The entity instance can be related to multiple instances of each other

Instruction:

- ***If "One" end of the relationship is owning side, then foreign key column is generated for the entity in database***
- ***If "Many" end of the relationship is the owning side, then join table is generated.***

Cascade Operation

The java.persistence.CascadeType enumerated type defines the cascade operation that are applied in the cascade element of the relationship annotations

Cascade Operations [Conti..]

Cascade Operation	Description
ALL	All cascade operations will be applied to the parent entities related entity. All is equivalent to specifying cascade={DETACH, MERGE,PERSIST, REFRESH, REMOVE}
DETACH	If the parent entity is detached from the persistence context, the related entity will also be detached
MERGE	If the parent entity is merged into the persistence context, the related entity will also be merged
PERSIST	if the parent entity is persisted into the persistence context, the related entity will also be persisted
REFRESH	if the parent entity is refreshed in the current persistence context, the related entity will be refreshed
REMOVE	if the parent entity is removed from the current persistence context, the related entity will also be removed

Cascade Operations [Conti..]

For Example:

A line item is part of an order, if the order is deleted, the line item also should be deleted using the cascade = REMOVE element specification for @OneToOne and @OneToMany relationships. This is called a cascade delete relationship.

```
@OneToMany(cascade = REMOVE, mappedBy = “employee”)
public Set<Name> getNames() { return names; }
```

Inheritance

- ***An important capability of the JPA is its support for inheritance and polymorphism***
- ***Entities can inherit from other entities and from non-entities.***

The @Inheritance annotation identifies a mapping strategy:

1. **SINGLE_TABLE**
2. **JOINED**
3. **TABLE_PER_CLASS**

Managing Entities

- ***Entities are managed by the entity manager, which is represented by java.persistence.EntityManager instance.***
- ***Each EntityManager instance is associated with a persistence context, which defines particular entity instance are created, persisted and removed.***

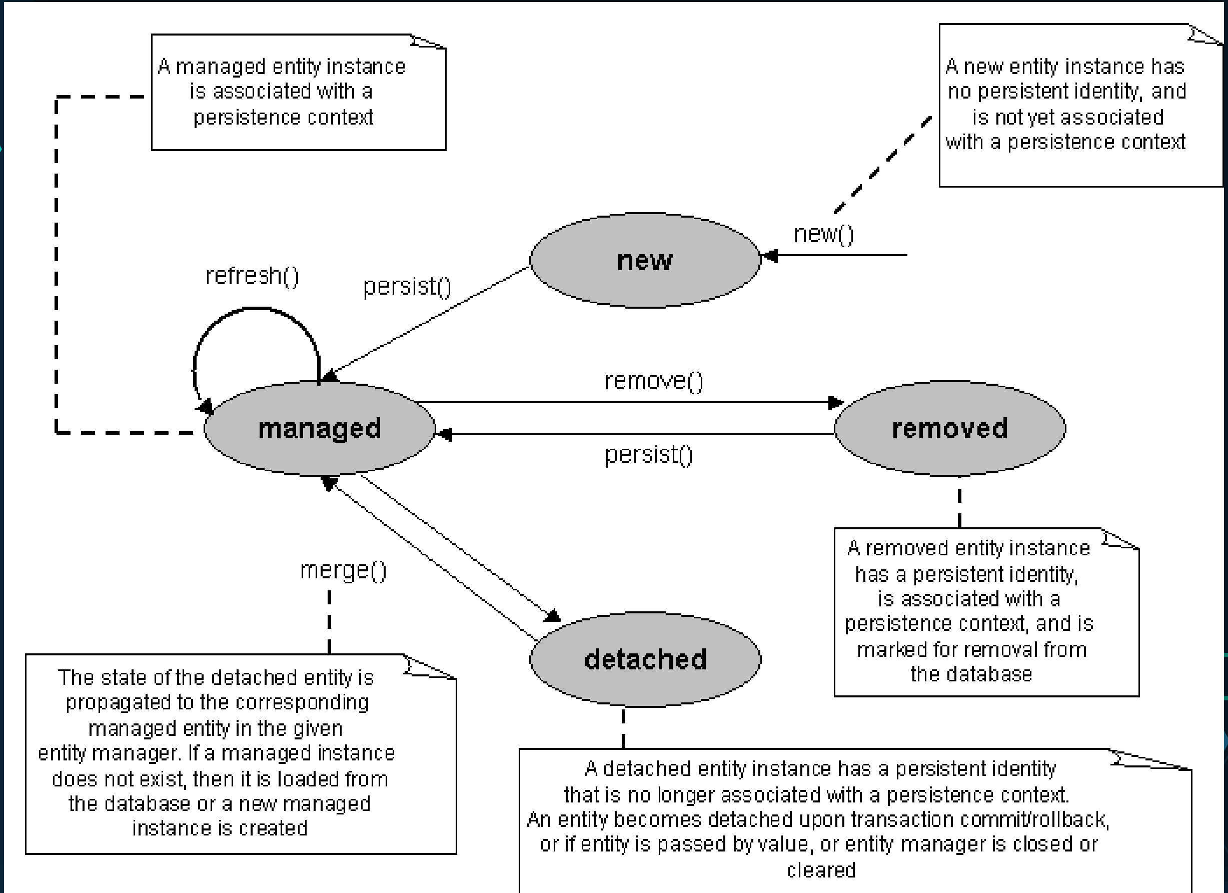
Managing Entities [Conti..]

To Obtain Entity Manager instance, you must first obtain an EntityManagerFactory instance by injecting it into the application component by means of the java.persistence.PersistenceUnit annotation.

The Example illustrates how to manage transactions in an application that uses an Application-Managed Entity Manager:

```
          @PersitenceContext
EntityManagerFactory emf;
EntityManager em;
@Resource,
UserTransaction utx;
...
em = emf.createEntityManager();
try {
  utx.begin();
  em.persist(SomeEntity);
  em.merge(AnotherEntity);
  em.remove(ThirdEntity);
  utx.commit();
} catch (Exception e) {
  utx.rollback();
}
```

Entity LifeCycle Management



A managed entity instance is associated with a persistence context

A new entity instance has no persistent identity, and is not yet associated with a persistence context

A removed entity instance has a persistent identity, is associated with a persistence context, and is marked for removal from the database

The state of the detached entity is propagated to the corresponding managed entity in the given entity manager. If a managed instance does not exist, then it is loaded from the database or a new managed instance is created

A detached entity instance has a persistent identity that is no longer associated with a persistence context. An entity becomes detached upon transaction commit/rollback, or if entity is passed by value, or entity manager is closed or cleared

API Operations on Entity

List of EntityManager API operations:

- **persist()**- *Insert the state of an entity into db*
- **remove()**- *Delete the entity state from the db*
- **refresh()**- *Reload the entity state from the db*
- **merge()**- *Synchronize the state of detached entity with the pc*
- **find()**- *Execute a simple PK query*
- **createQuery()**- *Create query instance using dynamic JP QL*
- **createNamedQuery()**- *Create instance for a predefined query*
- **createNativeQuery()**- *Create instance for an SQL query*
- **contains()**- *Determine if entity is managed by pc*
- **flush()**- *Force synchronization of pc to database*

Persistence Unit

A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence Unit [Conti..]

Persistence units are defined by the resource/META-INF/persistence.xml configuration file.

The following is an example persistence.xml file:

```
<persistence>
    <persistence-unit name="EmployeeManagement">
        <description>This unit manages employee and company.
            It does not rely on any vendor-specific features and can
            therefore be deployed to any persistence provider.
        </description>
        <jta-data-source>jdbc/EmployeeDB</jta-data-source>
        <jar-file>EmployeeApp.jar</jar-file>
        <class>com.Employee</class>
        <class>com.Company</class>
    </persistence-unit>
</persistence>
```

Hibernate Persistence [Conti..]

For Example:

```
<persistence-unit name="demoPU" transaction-type="RESOURCE_LOCAL">
     <provider>org.hibernate.ejb.HibernatePersistence</provider>
          <properties>
              <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect"/>
             <property name="javax.persistence.jdbc.driver"
value="org.postgresql.Driver" />
             <property name="javax.persistence.jdbc.url"
value="jdbc:postgresql://localhost:5432/ demo" />
            <property name="javax.persistence.jdbc.user" value="postgres"/>
            <property name="javax.persistence.jdbc.password" value="***"/>
            <property name="hibernate.hbm2ddl.auto" value="create" />
        </properties>
</persistence-unit>
```

Hibernate.hbm2ddl.auto

This hibernate,hbm2ddl.auto is used to validate or export DDL schema to the database when the SessionFactory is created.

Possible Values are:

- **validate**
- **create**
- **update**
- **create-drop**

Configuration

EntityManager injection:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="persistenceUnitName" value="demoPU" />
        <property name="jpaVendorAdapter">
            <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <property name="generateDdl" value="true" />
                <property name="showSql" value="true" />
                <property name="databasePlatform"
value="org.hibernate.dialect.PostgreSQLDialect" />
            </bean>
        </property>
        <property name="jpaProperties">
          <props>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
          </props>
        </property>
    </bean>
```

Spring Configuration

Transaction Manager injection:

```
<bean id="transactionManager“
        class=“org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory"       ref="entityManagerFactory"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>
```

Querying Entities

The Java Persistence API provides the following methods for querying entities:

- ***The Criteria API is used to create type safe queries using Java Programming language API's to query for entities and their relationships.***

- ***The Java Persistence query language (JPQL) is a simple, string-based language similar to SQL used to query entities and their relationships.***

Difference Between JPA and Hibernate

JPA is just guidelines to implement the Object Relational Mapping (ORM) and there is no underlying code for the implementation.Where as, Hibernate is the actual implementation of JPA guidelines. When hibernate implements the JPA specification.

Hibernate is a JPA provider. When there is new changes to the specification, hibernate would release its updated implementation for the JPA specification.

In summary, JPA is not an implementation, it will not provide any concrete functionality to your application. Its purpose is to provide a set of rules and guidelines that can be followed by JPA implementation vendors to create an ORM implementation in a standardized manner. Hibernate is the most popular JPA provider.

Resources

- Oracle: Java Persistence API:- http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html
- The Java Persistence API - A Simpler Programming Model for Entity Persistence:- http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html
- JPA Annotation Reference:- http://www.oracle.com/technetwork/middleware/ias/toplink-jpa-annotations-096251.html
- Hibernate Reference Documentation:- http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/